Search-based Testing for Embedded Telecommunication Software with Complex Input Structures: An Industrial Case Study

Kivanc Doganay[1,2], Sigrid Eldh[3,4], Wasif Afzal[2], and Markus Bohlin[1,2]

[1]SICS Swedish ICT AB, Kista, Sweden
[2]Mälardalen University, Västerås, Sweden
[3]Ericsson AB, Kista, Sweden
[4]Karlstad University, Karlstad, Sweden
kivanc@sics.se,
sigrid.eldh@ericsson.com,
wasif.afzal@mdh.se,
markus.bohlin@sics.se

SICS Technical Report: T2014:03

28 July 2014

Abstract

In this paper, we discuss the application of search-based software testing techniques for unit level testing of a real-world telecommunication middleware at Ericsson. Input data for the system under test consists of nested data structures, and includes non-trivial variables such as uninitialized pointers. Our current implementation analyzes the existing test cases to discover how to handle pointers, set global system parameters, and any other setup code that needs to run before the actual test case. Hill climbing (HC) and (1+1) evolutionary algorithm (EA) metaheuristic search algorithms are used to generate input data for branch coverage. We compare HC, (1+1)EA, and random search as a baseline of performance with respect to effectiveness, measured as branch coverage, and efficiency, measured as number of executions needed. Difficulties arising from the specialized execution environment and the adaptations for handling these problems are also discussed.

1 Introduction

Embedded systems are prevalent in many industries such as avionics, railways and telecommunication systems. The use of specialized microprocessors such as digital signal processors (DSPs), electronic control units (ECUs), and programmable logic controllers (PLCs) have enabled more complex software in the embedded domain. As a consequence, quality control has become more time consuming and costly, similar to the non-embedded software domains.

It is well known that software testing is an expensive activity [25, 6, 5]. Thus considerable research has focused on automating different test activities, notably software test data generation. In recent years, the use of metaheuristic search algorithms have shown promise in automating parts of software testing efforts [16]. The approach of using metaheuristic search techniques to automatically generate test data is commonly referred to as search-based software testing (SBST).

A recent systematic review by Ali et al. [2] found that the most common target of empirical studies in SBST is control-flow based coverage criteria, with the branch coverage receiving the most attention. While the use of branch coverage criteria might be maturing in SBST, the review by Ali et al. point out that scalability analyses are very rare in empirical SBST studies. While agreeing with Ali et al. [2], we believe that the scalability analysis of SBST research has to be investigated in parallel with industrial applicability. SBST has received ever increasing attention in the academia, but it is far from becoming a commonly known tool in industrial practice. While random testing and fuzzing have gained reasonable visibility and acceptance, search-based approaches are not yet adopted in the industry. To gain wider acceptance, we believe that experiments of search-based techniques on real-world industrial software should be performed. With a family of such experiments, we would be in a better position to argue for the industrial uptake of SBST.

In this paper, we discuss the application of search-based testing techniques for unit level testing of a real-world business critical telecommunication middleware at Ericsson. We use hill climbing (HC) and (1+1) evolutionary algorithm (EA) metaheuristic search algorithms to generate input data that exercise different branches in the control flow. Random search (RS) is used as a baseline of performance, and these three algorithms are compared against each other for effectiveness, measured as branch coverage, and efficiency, measured as number of executions. Input data for the system under test consists of nested data structures, and includes non-trivial variables such as uninitialized pointers. Our current implementation analyzes the existing test cases created by domain experts to decide how to handle pointers, or global systems parameters, among other things. We also discuss difficulties arising from the specialized execution environment, and the adaptations for handling these problems.

2 Background

Search-based software testing translates a given *testing task* into a search problem that various metaheuristic search algorithms can be applied. A common concrete testing task is generating test input data for structural coverage, such as statement coverage, branch coverage, and path coverage. Through usage of a fitness function crafted for the particular coverage criteria, the search algorithms are employed

to find test input that achieve the coverage goals. SBST has been successfully applied for various test data generation purposes, such as structural coverage [2], functional testing [23] and non-functional testing [1]. In this section we explain the search algorithms used in our study for test data generation for branch coverage.

2.1 Hill Climbing

Hill climbing (HC) is one of the fundamental metaheuristic local search algorithms. The algorithm starts from a random point in the search space and checks the *fitness* of neighboring solutions. If a neighbor has a better fitness, the search moves to that point and keeps iterating by again evaluating the neighbors of the new solution. If none of the neighbors are better than the current solution, it is common to restart the search from a random point in the search space. We follow the same restart strategy in our experiments.

There are certain variations of hill climbing, some with their own descriptive names. For example, *steepest descent* (or *ascent*) checks every neighbor and chooses the one with the best fitness. In our experiments we use hill climbing with alternating variable method, where the search continues in the direction of the first improving neighbor, by using increasing step sizes [12].

For the problem of test input generation, a candidate solution (or a point in the search space) is represented as a vector of values. The neighborhood is the set of new solutions that result from a small change to one of the elements of the current candidate solution. Each candidate solution is executed against the test subject, and a *fitness* value is calculated. In our implementation, we use the popular fitness function for branch coverage that combines *approach level* and *branch distance* [22].

2.2 (1+1)EA

(1+1)EA is an evolutionary algorithm that uses a single individual instead of a population. The parent solution is mutated at every iteration, to produce one offspring. If the offspring has a better or equal fitness, it replaces the parent. Otherwise the original parent is kept. When mutating the current parent solution consisting of n variables, each variable typically has a $1/n$ chance of being mutated. Therefore, in (1+1) EA there is a non-zero probability to reach any state from any state, and hence it can be classified as a *global* search algorithm. The process is repeated at every iteration until a goal solution is found (i.e., targeted branch is covered) or a stopping condition such as an execution limit is reached. Fitness of the offspring solution at each iteration is calculated using the same fitness function as HC.

2.3 Random Search

Random search is a straightforward algorithm commonly used as a base line for performance comparisons. For each element in the input vector, a valid value is randomly sampled using a uniform probability for the possible values. This process is repeated until a goal solution is found or a stopping condition is reached.

3 System Under Test

The system under test is written in the DSP-C programming language. DSP-C is an extension of the C language with additional types and memory quantifiers designed to allow programs to utilize hardware architectural features of digital signal processors (DSP), for example fixed point arithmetic [14]. Ericsson uses a proprietary compiler that compiles DSP-C code for various proprietary DSP architectures. There is a corresponding simulator (also built in-house by Ericsson) capable of executing the resulting binaries in a normal Linux environment. The whole tool chain is connected via a test execution framework, again specific to Ericsson. Currently we do not integrate to the test framework, but we extract certain information from it, after which the SBST algorithms are run as a separate tool.

3.1 Analysis and Instrumentation

We adapted `pycparser`[1], which produces the abstract syntax tree (AST) for the C language, to the DSP-C language including Ericsson specific memory identifiers. We chose `pycparser` as it is a minimalistic parser that is relatively easy to understand and modify, compared to full fledged frameworks for analyzing the C language. Our tool analyzes the resulting AST to produce the control flow graph (CFG) of the function under test (FUT).

The second step is to instrument the FUT by inserting observation code at the branching points in the CFG (e.g., `if` statements). The code is instrumented without changing its functional behavior using observation functions (`observe_*`), which save the *observation identifier* and the observed value in a circular buffer. For example, the statement `if(a>b && c)` is instrumented as `if(observe_gt(12,0,a,b) && observe(12,1,c))`, where 12 is the branch identifier in the CFG, while 0 and 1 are the clause identifiers in the given expression. Note that the variable `c` will not be read if the first condition `(a>b)` is false. This ensures that instrumentation do not change the functional behavior of the FUT.

3.1.1 Existing test cases.

We also analyze the existing test cases (developed by Ericsson engineers) for a given FUT to discover the input interface, and any setup code that needs to execute before calling the FUT. Test cases are written as DSP-C functions that make multiple calls to the target FUT. Hence, a single tester function can be seen as multiple test cases, possibly with a common setup code. Ericsson uses an in-house test execution framework that maps test functions to FUTs using XML files. We parse these XML files to automatically discover the test functions. Alternatively, the user may explicitly point out the location of the test functions for a given FUT.

Each test function starts with variable declarations and possibly some *setup code*, such as allocating memory for a pointer, setting memory to null, or setting the value of a global variable. We replicate these setup code sections in the generated template test case. Then all assignments in the test code are parsed, and the discovered assignments are used to define the input space. Of these,

[1] Available at https://github.com/eliben/pycparser

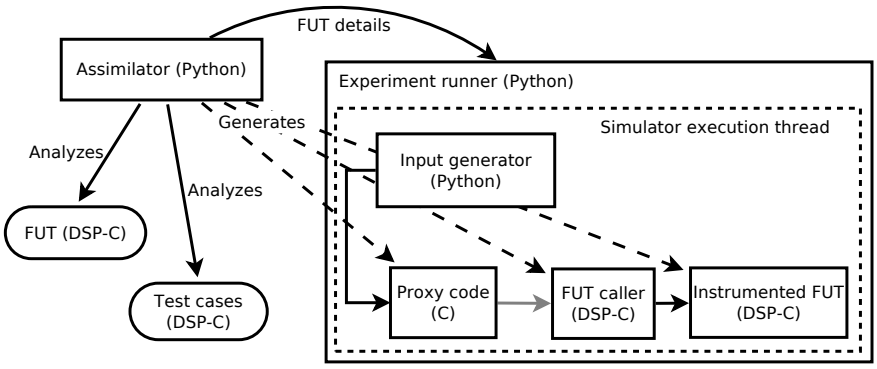

Figure 1: Prototype's architecture and the execution environment.

only the variables and fields of data structures which are set in at least one test function are used to construct the input space, as it is likely that variables and fields which are never set are uninteresting for testing purposes. In particular, this is true for variables which do not affect the execution of the FUT (e.g., output variables), or for variables which are initialized in the setup phase and should not be changed due to the logic of the system.

In our case, parsing the test code helps the analysis in two major ways. Firstly, we are able to handle complex/dynamic input structures, pointers, and variable length arrays. Secondly, the input space is reduced due to omission of structure fields that are not used in any existing test function. We also expect that omitting the unused input variables would make the resulting input vectors more human readable. However this idea is not part of the case study, and not yet evaluated.

3.2 Execution Environment

In our study we used Ericsson's proprietary DSP simulator. The simulator allows C code (compiled by `gcc`) to interact with the simulated DSP-C code. However, it is not possible a make direct function calls. Instead, the C code asks the simulator to continue the execution with the particular DSP-C function. We use a shared common memory area to pass arguments from the C code to the DSP-C function. The simulator has control over both the simulated DSP-C code and the normal C code. A significant consequence is observed when the DSP-C code throws an explicit exception or tries an invalid/unsafe operation (e.g., reading unallocated memory). In such cases, the simulator halts the execution without giving the C code the chance to read the observations that were collected up to the point where the exception occurred. Therefore, the state of the search algorithms are continuously saved to disk, so that the search can recover to the last pre-crash state and continue to run by avoiding the input that led to the crash.

The above mentioned interactions led us to heavily adapt our prototype for the simulator. Main modules and their relations are depicted in Fig. 1. The `Assimilator` analyzes the FUT and existing test cases for the FUT, as discussed in Section 3.1, creates an instrumented version of the FUT, and saves any information needed at the dynamic test generation step (e.g., branches to be

Table 1: Structural information of the functions that are included in our study.

Function under test	LOC	Number of branches	Input vector size	
			Original	Reduced
`Func_A`	191	20	508	30
`Func_subA`	37	6	508	30
`Func_B`	352	20	143	28
`Func_C`	179	32	659	146

targeted in the CFG). The `Input generator` implements the search algorithms using a vector representation of basic types, which is oblivious to the actual input structures (arrays, pointers, nested C structs, etc.). The `Proxy code` and `FUT caller` are synthesized automatically after the analysis of existing test cases by the `Assimilator` module. The `Proxy code` passes dynamically calculated input values to the simulated DSP-C environment. `FUT caller` is the *template test case* that includes any setup code (as discussed in Section 3.1), input packing from a vector representation into complex structures, and a single call to the FUT.

After the FUT execution ends, the C code reads the observation data that is collected during the FUT execution. Then, the `Input generator` module calculates the fitness values using the observed data and continues with the particular search algorithm. However, if there is an exception during the FUT execution, the simulator immediately stops the execution without allowing the observation data to be collected. A simple Python script (shown as `Experiment runner` in Fig. 1) monitors and restarts the simulator after such interruptions. The `Input generator` module continuously saves enough information to the disk so that it is able to recover from a crash, and continue the search algorithm by skipping the input vector that caused the exception.

4 Experimental Evaluation

In this section we compare the experimental results of random testing, hill climbing, and (1+1)EA algorithm for branch coverage on a set of four functions in a library sub-module of Ericsson's middleware code base. The particular sub-module was selected as it was considered suitable to be tested using the simulator, rather than the real hardware. In particular, suitable sub-modules should have no or minimal inter-module dependencies, and should be without timing critical requirements.

The functions with the most lines of code and branching statements were selected. The chosen FUTs are listed in Table 1 with basic structural information. `Func_A`, `Func_B`, and `Func_C` are directly tested by the existing unit tests. However, `Func_subA` is not executed directly but through `Func_A`, which we mimic in our test data generation approach. Therefore, `Func_A` and `Func_subA` share the same input vector.

Experiments were run on Ericsson servers using the proprietary simulator. We use execution count (i.e., the number of calls to the FUT) for comparison, instead of the clock time, so that secondary concerns such as efficiency of our implementation, interactions with the simulator, or the server load do not affect the results. For practical purposes we imposed a limit of 1000 executions (FUT

Table 2: Branch coverage achieved by hill climbing, (1+1)EA, and random search algorithms, as well as the existing test cases.

FUT	Exist.	Hill Climbing				(1+1)EA				Random Search			
		min	mean	med.	max	min	mean	med.	max	min	mean	med.	max
`Func_A`	1.0	0.95	0.996	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
`Func_subA`	0.83	0.5	0.99	1.0	1.0	0.5	0.983	1.0	1.0	0.5	0.947	1.0	1.0
`Func_B`	0.85	0.95	0.998	1.0	1.0	0.7	0.701	0.70	0.75	0.7	0.705	0.70	0.75
`Func_C`	1.0	0.656	0.746	0.75	0.812	0.75	0.812	0.812	0.875	0.75	0.824	0.812	0.906

Table 3: Number of executions (fitness evaluations or FUT calls) that each search algorithm needed before terminating.

FUT	Hill Climbing				(1+1)EA				Random Search			
	min	mean	med.	max	min	mean	med.	max	min	mean	med.	max
`Func_A`	330	641	576	1555	288	551	506	1059	73	151	150	310
`Func_subA`	167	448	382	3234	110	493	408	3143	52	436	131	3034
`Func_B`	1853	2231	2194	2838	5650	6401	6364	6955	5382	5996	6042	6070
`Func_C`	10043	11722	11551	13763	7710	9396	9256	11970	5605	7353	7591	9686

calls) per targeted branch. In order to account for the stochastic nature of the search algorithms, we repeated each experiment 50 times per algorithm per FUT. The total time for executing all experiments was around 6 days.

Resulting coverage and execution counts of the algorithms per FUT are compared using the Vargha-Delaney effect size measure ($\hat{A}_{12}$), which is a non-parametric statistic that do not assume a particular distribution of data. We also compute the p-values for the statistical significance using Mann-Whitney U measure, which is also a non-parametric test.

4.1 Results

The minimum, maximum, mean and median values for the achieved branch coverage (during 50 independent runs) by each algorithm are shown in Table 2. Branch coverage rate of the existing test cases are also listed for comparison. Figure 2 shows the same coverage values in the form of box plots. For `Func_A` and `Func_subA` all algorithms were able to reach full coverage at most of the runs, while some branches of `Func_B` were covered only by the hill climbing algorithm.

Table 3 shows the number of executions (FUT calls) needed per algorithm for each FUT. Again the minimum, mean, median, and maximum values among the 50 independent runs are reported in the table. Box plots of the same values are shown in Fig. 3.

The Vargha-Delaney effect size measures ($\hat{A}_{12}$) for pairwise comparison of the algorithms with respect to coverage is shown in Table 4. $\hat{A}_{12}$ values close to 0.5 indicate that one algorithm is not clearly better than the other (or the difference is small), while high values favor the first named algorithm and low values indicate the second algorithm is better. For example, *HC vs. RS* for `Func_B` is high (1.0) – which indicates that hill climbing algorithm performed

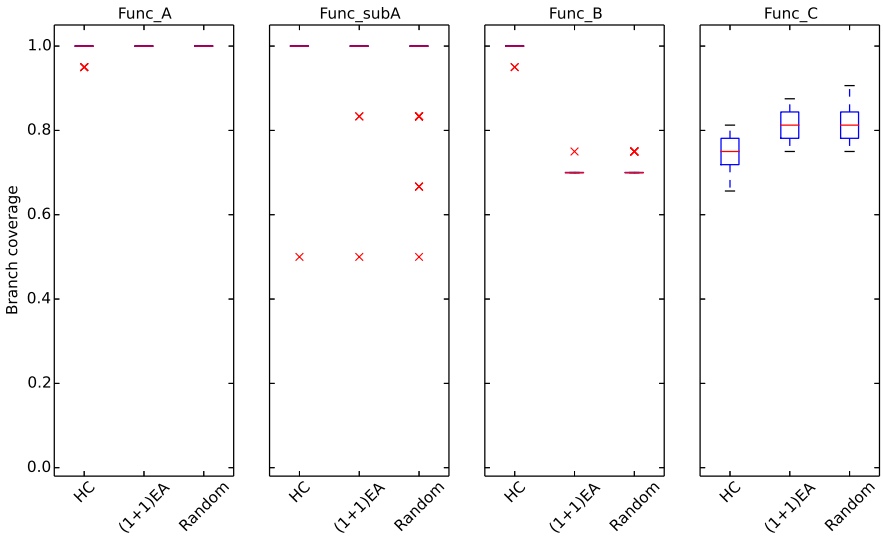

Figure 2: Box plots of branch coverage distribution of each algorithm per FUT. The data is also shown in Table 2.

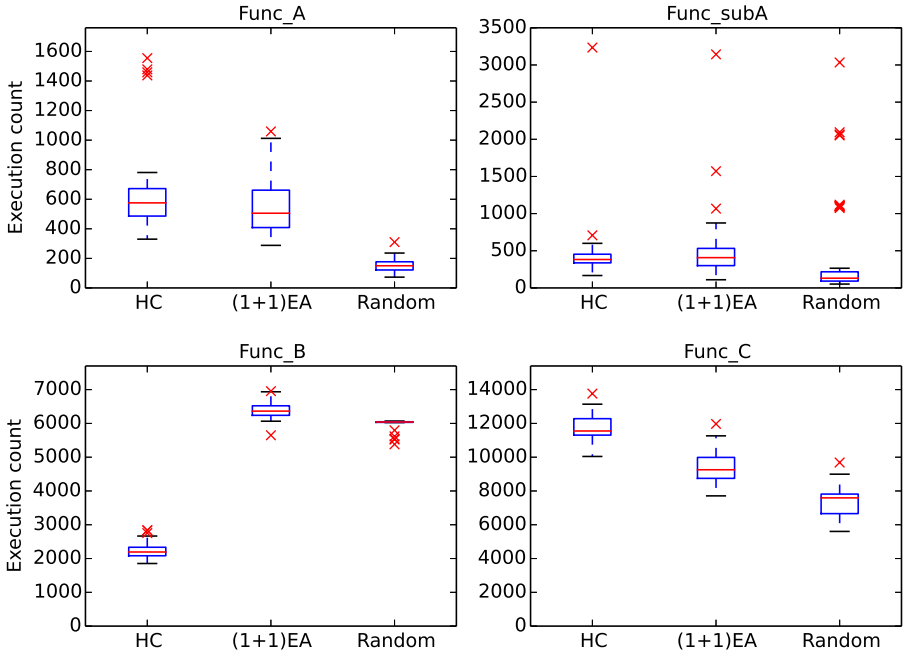

Figure 3: Box plots of execution count distribution of each algorithm per FUT. The data is also shown in Table 3.

Table 4: Vargha-Delaney $\hat{A}_{12}$ statistics and p-values for pairwise comparison of coverage achieved by each algorithm per FUT.

FUT	HC vs. RS		(1+1)EA vs. RS		HC vs. (1+1)EA	
	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value
`Func_A`	0.46	0.0216	0.5	0.0*	0.46	0.0216
`Func_subA`	0.598	0.0014	0.58	0.0116	0.52	0.162
`Func_B`	1.0	2.68×10^{-22}	0.46	0.0477	1.0	5.37×10^{-23}
`Func_C`	0.0738	4.37×10^{-9}	0.422	0.142	0.086	1.73×10^{-13}

Table 5: Vargha-Delaney $\hat{A}_{12}$ statistics and p-values for pairwise comparison of execution (FUT calls or fitness evaluations) needed by each algorithm per FUT.

FUT	HC vs. RS		(1+1)EA vs. RS		HC vs. (1+1)EA	
	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value
`Func_A`	0.0	3.53×10^{-18}	0.0012	4.23×10^{-18}	0.39	0.029
`Func_subA`	0.221	7.86×10^{-7}	0.236	2.82×10^{-6}	0.534	0.279
`Func_B`	1.0	3.51×10^{-18}	0.0196	6.33×10^{-17}	1.0	3.53×10^{-18}
`Func_C`	0.0	1.95×10^{-11}	0.0962	4.76×10^{-8}	0.0336	4.69×10^{-16}

better than random search with a strong effect size, for that particular FUT.

Together with the effect sizes ($\hat{A}_{12}$), p-values of Mann-Whitney U test are also reported in Table 4. Given p-values indicate the probability of reaching particular $\hat{A}_{12}$ value by chance, rather than a real effect. For instance, the high effect size observed while comparing hill climbing to random search for `Func_B` has an extremely small (2.68×10^{-22}) risk of being achieved by chance.

The Vargha-Delaney effect sizes for pairwise comparison of each algorithm's execution count are also calculated and reported in Table 5. Again, a high $\hat{A}_{12}$ value indicate that the first algorithm is favorable, though in this case a lower execution count is better. Calculated p-values are also reported in the table.

5 Discussion

5.1 Discussion on the Results

The results in Section 4.1 indicate that both HC, (1+1)EA, and RS were able to achieve high branch coverage most of the time (Fig. 2). At least one of the algorithms were able to achieve 100 % branch coverage in the case of three out of the four FUTs. For two of the FUTs (`Func_subA` and `Func_B`) the achieved branch coverage was higher than branch coverage of the existing test cases (Table 2). Therefore, we were able to increase the branch coverage for the FUTs that are studied in our case study. However, search algorithms did not always reach the highest possible branch coverage (`Func_C` in Table 2).

We observed that RS was not worse than other algorithms at branch coverage, except for `Func_B` where HC was the clear winner. This indicates that many of the branches are relatively easy to cover. One such branch predicate that we

found to be common in the code base is inequality comparison with a signed input value and a small constant, such as ($x \leq 10$). For example, a signed 32-bit integer input x would mean that the probability of a random input leading to the false and true branches approximately equal ($P(false) \approx P(true) \approx 0.5$). Similar situation can be observed if the variable being compared is unsigned type, but is defined as a bit fields with, e.g., 4-bit length. Such a variable would have a range of 0–15, which means that the randomly selected values have good chance of hitting the less likely branch as well ($P(false) = 5/15 = 0.33$). In embedded software bit fields are commonly used, due to the need of limiting the memory usage as much as possible.

Furthermore, on average RS is faster (i.e., needs less number of FUT calls) to cover a branch, if it can, compared to other algorithms (Fig. 3). It is known in the literature that RS is typically faster at covering easy branches than more informed search algorithms.

5.2 Discussion on the Case Study

Throughout the implementation of this industrial case study we have encountered various problems. We were not familiar with the DSP-C language which prevented us from using various toolboxes available for C, and led us to spend more time on implementation of the code analysis part. Another major implementation work involved the adaptations needed for the execution model of the simulator (e.g, indirect calls to FUTs and inability to catch exceptions as explained in Section 3.2). This can be seen as a special version of the *execution environment problem*, which usually refers to the concerns about interacting with the file system, the network, or a database in the context of non-embedded systems [17].

Another major point in the case study is that we did not have enough detailed technical knowledge of the system under test to understand the input space. To overcome this problem, we used existing test cases to automatically craft a test case template (Section 3.1). Furthermore, we reduced the input vectors by ignoring the input data structure fields that are not used by any of the existing test cases, which decreases the input size significantly (Table 1). It is not guaranteed that existing test cases include enough input fields to achieve full branch coverage – in other words, this is a speculative approach. However, results were promising, and we were able to achieve full coverage for most of the FUTs.

5.3 Threats to Validity

Due to practical reasons (such as computational resources) we imposed arbitrary limits on the execution of the algorithms on each targeted branch. Different execution limits might have led to differing results.

We selected limited number of FUTs from one sub-module. FUTs were selected among the functions with more lines of code and number of branches. We do not know if many smaller FUTs, instead of few big ones, would give significantly different results. In the future, we plan to extend this work to more sub-modules and functions in the code base.

6 Related Work

Search-Based Software Testing (SBST) has received much interest in recent years and a number of survey papers covering the field have been published [16, 1, 2, 9]. One of the most widely studied goal in SBST is to search for test data for full branch coverage of a program [9]. The seminal paper by Miller and Spooner [19] suggested using optimization techniques for satisfying a series of conditions that must be satisfied for a path to be executed. Korel [12] was one of the first authors to use a local search technique (the alternating hill climbing method) for branch coverage. Xanthathis [24] proposed the application of a global search technique (genetic algorithm) for test data generation. Many other authors have since then applied search-based techniques for branch coverage [18, 20, 22]. Most of the work on search-based test data generation for structural testing consider the input to the program to be a fixed-length vector of input values, making the search space well-defined and fixed-size. This is not the case when inputs to a program are complex dynamic data structures or pointers. According to Baars et al. [4], problematic language constructs, such as variable length argument functions, pointers and complex data types (variable size arrays, recursive types such as lists, trees, and graphs) remain largely unsupported in SBST.

Alshraideh and Botacci [3] used a genetic algorithm to generate test data for covering branches with string predicates. Lakhotia et al. [13] combined search-based testing and symbolic execution [11] to generate test data for pointer inputs. They used an alternating variable hill climbing method with a set of constraint solving rules. The problem of path explosion was solved using an adaptation of a lazy initialization approach for dynamic data structures.

Symbolic execution is also the basis for concolic testing [8, 21]. Concolic testing uses symbolic execution to solve non-linear constraints by combining concrete execution with symbolic generation of path conditions. A path condition is constructed in symbolic execution which is a system of constraints in terms of the input variables describing when the path will be executed. The path condition can become unsolvable if it contains floating point variables or non-linear constraints. Inkumsah and Xie [10] combined SBST and concolic testing in a framework, targeting coverage of Java classes. Other examples on the use of concolic and SBST can be found in [7, 15].

7 Conclusion and Future Work

In this paper, we discussed a case study on application of SBST techniques for an industrial embedded software. During the experiments we were able to increase the total branch coverage of existing test cases. So we can say that the employed SBST techniques indicate beneficial results. However, we observed that randomly generated inputs were as effective as more informed search algorithms in many (but not all) cases, which indicate that there are many branches that are easy to cover.

Various difficulties were encountered at the implementation phase due to the specialized execution environment of the embedded software. Therefore, our prototype implementation was highly adapted to the system under test, instead of following a more generic structure. Another important characteristic of the system under test is its complexity and the specialized domain. Due to this, it

becomes difficult to define the input space of a FUT in the presence of non-basic data types (such as pointers). We alleviated this problem by generating test case templates from existing test cases for a given FUT.

For future work, we would like to extend the case study to more FUTs in the same or similar code base. We plan to investigate alternative ways of interacting with the simulator to reduce the execution times, which was a practical bottleneck for the experiments. Another interesting idea is the use of hybrid search algorithms that can cover easy branches swiftly (e.g., by using random search) and then move to other branches.

8 Acknowledgments

This work was supported by VINNOVA grant 2011-01377. Kivanc Doganay is supported by the Knowledge Foundation (KKS) through the ITS-EASY Industrial Research School in Embedded Software and Systems at Mälardalen University, Sweden.

We would like to thank Jonas Allander, Catrin Granbom, John Nilsson, and Andreas Ermedahl at Ericsson AB for their help in enabling the case study.

References

[1] Afzal, W., Torkar, R., Feldt, R.: A systematic review of search-based testing for non-functional system properties. Information and Software Technology 51, 957–976 (Jun 2009)

[2] Ali, S., Briand, L., Hemmati, H., Panesar-Walawege, R.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Transactions on Software Engineering 36(6), 742–762 (2010)

[3] Alshraideh, M., Bottaci, L.: Search-based software test data generation for string data using program-specific search operators. Software Testing, Verification & Reliability 16(3), 175–203 (2006)

[4] Baars, A., Lakhotia, K., Vos, T.E.J., Wegener, J.: Search-based testing, the underlying engine of future internet testing. In: Proceedings of the 2011 Federated Conference on Computer Science and Information Systems (FedCSIS'11) (2011)

[5] Beizer, B.: Software testing techniques. International Thomson Computer Press (1990)

[6] Bertolino, A.: Software testing forever: Old and new processes and techniques for validating today's applications. In: Jedlitschka, A., Salo, O. (eds.) Product-Focused Software Process Improvement, Lecture Notes in Computer Science, vol. 5089, pp. 1–1. Springer Berlin Heidelberg (2008)

[7] Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08). IEEE Computer Society, Washington, DC, USA (2008)

[8] Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. SIGPLAN Notes 40(6), 213–223 (2005)

[9] Harman, M., Mansouri, A., Zhang, Y.: Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Tech. Rep. TR-09-03, Department of Computer Science, King's College London (April 2009)

[10] Inkumsah, K., Xie, T.: Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'07). ACM, New York, NY, USA (2007)

[11] King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)

[12] Korel, B.: Automated software test data generation. IEEE Transactions on Software Engineering 16(8), 870–879 (1990)

[13] Lakhotia, K., Harman, M., McMinn, P.: Handling dynamic data structures in search based testing. In: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO'08). ACM, New York, NY, USA (2008)

[14] Leary, K., Waddington, W.: DSP/C: a standard high level language for DSP and numeric processing. In: International Conference on Acoustics, Speech, and Signal Processing, 1990. ICASSP-90. pp. 1065–1068 vol.2 (1990)

[15] Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proceedings of the 29th International Conference on Software Engineering (ICSE'07). IEEE Computer Society, Washington, DC, USA (2007)

[16] McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14(2), 105–156 (2004)

[17] McMinn, P.: Search-based software testing: Past, present and future. p. 153–163. ICSTW '11, IEEE Computer Society, Washington, DC, USA (2011)

[18] Michael, C.C., McGraw, G., Schatz, M.A.: Generating software test data by evolution. IEEE Transactions on Software Engineering 27(12), 1085–1110 (2001)

[19] Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Transactions on Software Engineering 2(3), 223–226 (1976)

[20] Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-data generation using genetic algorithms. Software Testing, Verification and Reliability 9(4), 263–282 (1999)

[21] Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. SIGSOFT Software Engineering Notes 30(5), 263–272 (2005)

[22] Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology 43(14), 841 – 854 (2001)

[23] Wegener, J., Bühler, O.: Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In: In Proceedings of GECCO. pp. 1400–1412. Springer-Verlag (2004)

[24] Xanthakis, S., Ellis, C., Skourlas, C., Gall, A.L., Katsikas, S., Karapoulios, K.: Application of genetic algorithms to software testing (application des algorithmes gènètiques au test des logiciels). In: Proceedings of the 5th International Conference on Software Engineering and its Applications. pp. 625–636 (1992)

[25] Yang, B., Hu, H., Jia, L.: A study of uncertainty in software cost and its impact on optimal software release time. IEEE Transactions on Software Engineering 34(6), 813–825 (2008)